A Formal Model Of SystemC Components Using Fractal Hypergraphs

Nicolas Vallée Bruno Monsuez *

Abstract— **In this paper, we introduce a new mathematical structure: *fractal hypergraph*. Due to the hierarchical and compositional nature of *fractal hypergraphs*, representations based on *fractal hypergraphs* can capture and abstract the object-oriented nature of SystemC. We propose a formal semantics of SystemC components based on *fractal hypergraphs* that has already be used in a formal debugger of SystemC components.**

Keywords: SystemC, Formal semantics, Hierarchical Models, Hypergraphs

1 Introduction

SystemC becomes a popular language for modeling complex hardware systems. Compared with other hardware description languages, SystemC is more feasible for designing large-scaled systems and modeling high level behaviors.

While a major goal of SystemC is to enable system verification at an higher level of abstraction, formal verification of SystemC design is still in its infancy. The difficulty to statically analyze and verify SystemC comes from the object-oriented nature of SystemC that supports hierarchy, modularity and parametricy as well as from its sophisticated event driven simulation semantics.

If we focus on the object-oriented nature of SystemC, we can identify the following key aspects of SystemC that should be captured and fully supported by the formal representation of SystemC components in a formal verification tools to provide a valuable verdict.

- The formal representation copes with different abstraction levels; it also must support the refinement.
- The formal representation must support program code as well as hardware description; a semantics for program execution and a semantics for system simulation must be provided.
- The formal representation must support assembling modular and parametrized components.

In this paper, we propose a new hierarchical model based on hypergraphs that can represent SystemC components and that provide the adequate constructions to capture the object-oriented nature of SystemC components. We also define a trace-based semantics based on hypergraphs and show how this hypergraph based semantics is adequate to represent SystemC components. We finally show how object-oriented and component-oriented operations like connecting a sub-component to a main component, instanciating a component with respect to type or value parameters can easily be expressed with this semantics.

The paper is organized as follow; section 2 gives the mathematical definition of the hypergraphs model: *fractal hypergraphs* and introduces the underlying trace based semantics; section 3 gives the semantics of SystemC components and finally section 4 presents how fundamental concepts like the compositional approach are nicely captured by this *fractal hypergraphs* based semantics.

2 A Fractal hypergraph based semantics

Graphs are certainly one of the simplest and most universal model for a large variety of systems including hardware models as well as programs. If graphs define the structure of the model, graph transformation can be exploited to explain how the model is built (compilation or synthesis of the system) and how the model evolves (computation) of the system.

However, when we want to represent complex circuits or systems as well as complex object-oriented system description, the absence of hierarchy [1] is certainly one of the main default of graph-based representations. To overcome the limitation of graphs, we introduce a new mathematical extension of graphs called *fractal hypergraphs*.

2.1 Fractal Hypergraphs

Hypergraphs [2] are a mathematical extension of graphs: A hypergraph is a graph whose hyperedges may connect two or more vertices. Directed hypergraphs [3] are hypergraphs where the hyperedges connects a set of one or more vertices to another set of one or more vertices. Like in standard directed graphs where the directed edges of a graph modelize a transition between an initial state and a final state, the directed hyperedges of a hypergraph modelize one or more transition that connects a set of initial states to a set of final states.

With respect to this approach, the hyperedges denote any system that connects a set of initial states to a set of final states. An interesting subset of those ones are the

*École Nationale Supérieure de Techniques Avancées, UEI, 32 Bd Victor, 75739 Paris cedex 15, France, FirstName.LastName@ensta.fr

systems that can also be represented using a graph. For instance, automata's and other graph based representation like petri nets define a transition system that maps a set of initial states to a set of final states. Those transition systems could be abstracted by a hyperedge that maps the initial states to the final states. We would like to generalize this notion to hypergraphs, restricting the hyperedges to hypergraphs. To quickly summarize, *fractal hypergraphs* [4, 5] are hypergraphs where the hyperedges between the vertices are defined by hypergraphs.

Definition 1 (Basic fractal hypergraph)
A basic fractal hypergraph $\mathscr{HB}$ is defined as:

- *a set of vertices $\mathscr{V}$,*
- *a set of sub-fractal hypergraphs $\mathscr{H} = (h_i = (\mathscr{V}_i, \mathscr{H}_i, {}^{in}\mathscr{V}_i, {}^{out}\mathscr{V}_i, {}^{in}\partial_i, {}^{out}\partial_i, {}^{in}\mathscr{E}_i, {}^{out}\mathscr{E}_i, \mathscr{E}_i))_{i \in I}$*
- *a set of edges ${}^{in}\mathscr{E}$ – ie. a binary relation ${}^{in}\mathscr{E} \in \wp(\mathscr{V} \times (\bigcup_i {}^{in}\mathscr{V}_i))$ – whose every edge connects a vertex $v \in \mathscr{V}$ to an entry vertex ${}^{in}v_n \in {}^{in}\mathscr{V}_n$ of the sub-fractal hypergraph h_n,*
- *a set of edges ${}^{out}\mathscr{E}$ – ie. a binary relation ${}^{out}\mathscr{E} \in \wp((\bigcup_i {}^{out}\mathscr{V}_i) \times \mathscr{V})$ – whose every edge connects an exit vertex ${}^{out}v_n \in {}^{out}\mathscr{V}_n$ of the sub-fractal hypergraph h_n to a vertex $v \in \mathscr{V}$,*
- *a set of edges $\mathscr{E}$ - ie. a binary relation $\mathscr{E} \in \wp(\mathscr{V} \times \mathscr{V})$ - each edge connects a vertex v_o to another one v_d*

Definition 2 (Fractal hypergraph)
A fractal hypergraph is defined as :

- *a basic fractal hypergraph $\mathscr{HB} = (\mathscr{V}, \mathscr{H}, {}^{in}\mathscr{E}, {}^{out}\mathscr{E}, \mathscr{E})$*
- *a set of entry vertices ${}^{in}\mathscr{V}$, a set of exit vertices ${}^{out}\mathscr{V}$,*
- *a set of entry edges ${}^{in}\partial$ – ie. a binary relation ${}^{in}\partial \in \wp({}^{in}\mathscr{V} \times \mathscr{V})$ that connects the entry vertices ${}^{in}\mathscr{V}$ to vertices of the basic fractal hypergraph $\mathscr{V}$*
- *a set of entry edges ${}^{out}\partial$ – ie. a binary relation ${}^{out}\partial \in \wp({}^{out}\mathscr{V} \times \mathscr{V})$ connecting the entry vertices ${}^{in}\mathscr{V}$ to vertices of the basic fractal hypergraph $\mathscr{V}$*

2.2 Interesting properties of the model

Using fractal hypergraph as representation of SystemC components allow to capture among others the following key aspects of modular and parametrized components.

2.2.1 Multiple hierarchy

A hyperedge corresponds to a given abstraction level. Hyperedges may contain an embedded fractal hypergraph; each hyperedge of this embedded fractal hypergraph can express a different abstraction level. Refining a behavior is achieved by substituting a simple hyperedge with a hyperedge that embeds a fractal hypergraph.

Each hyperedge also plays the role of an abstract interface of a sub-component. Sub-components are modeled by fractal hypergraphs. The model supports substituting a component with another component; a fractal hypergraph that models a component may be replaced by another fractal hypergraph that represents another component, the only requirement is that both fractal hypergraphs share the same abstract interface (*i.e.* the same set of initial and final states) and consequently the same abstract properties.

2.2.2 Concurrency

A hyperedge $\mathscr{E}$ that connects a set of initial states to a set of final states may encapsulate two or more concurrent transitions, each of those transitions are represented by a sub-hyperedge $\mathscr{E}_i^{\text{sub}}$ that is located in the fractal sub-hypergraph that describes the hyperedge $\mathscr{E}$. The concurrent transitions denoted by the sub-hyperedges $\mathscr{E}_i^{\text{sub}}$ can generate events or can be activated by some signal are triggered; this is exposed in 3.

2.2.3 Aggregation & Parametrization

Components are represented by fractal hypergraphs. Components may statically aggregates sub-components, each sub-component is also represented by a sub-fractal hypergraph; section 4 describes the binding between components and sub-components. Components may also be parametrized by sub-components or values. In this case, the component will be represented by a fractal hypergraph $\mathscr{H}$ and each time a parameter is required, an empty sub-hypergraph $\mathscr{H}_i^{\text{param}}$ is inserted in the fractal hypergraph $\mathscr{H}$. When instantiating the parametrized fractal hypergraph $\mathscr{H}$, the empty sub-hypergraphs $\mathscr{H}_i^{\text{param}}$ get replaced by the hypergraphs that denote the selected implementation; the process is described in section 3.

2.3 A semantics based on fractal-hypergraphs

2.3.1 Defining an underlying semantics for fractal hypergraphs

Fractal hypergraphs are the model for system representation. Their structure and their transformations define an underlying semantics. This semantics is a *storeless trace-based semantics*, which enables processing both symbolic execution and static analysis.

A trace-based semantics [6, 7] manipulates paths representing the execution traces. The traces contains the history of the symbolic execution or static analysis ; it totally defines the current context.

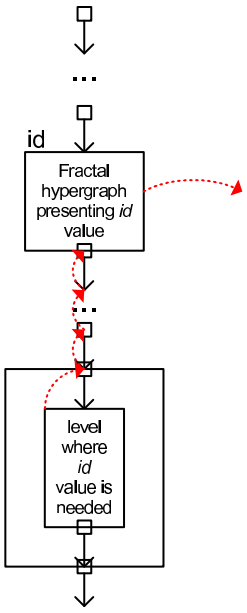
Storeless semantics [8] do not require any external structures. Storeless semantics also avoid to manage both environment and system representation. Figure 1 describes

how a value embedded into a fractal hypergraph, which is associated to an identifier. Using a storeless semantics instead of a classical semantics reduces the number of fastidious manipulations that occur when handling some advanced constructions; like exceptions in denotational semantics [9] or closures in operational semantics [10] as well as concurrent executions.

Fractal hypergraphs represent the execution traces. In fact symbolic execution unroll the fractal hypergraph. At each step, the structure of fractal hypergraphs represents all the history of the execution.

Fractal hypergraphs represent values, function closures, as well as objects. An evaluation of such an fractal hypergraph returns whatever value is associated.

To summarize, fractal hypergraphs provide the values of variables, the closures of functions, the structures and the instances of classes and the execution traces in the meantime. Merging data and code is a key idea of object-oriented design. Fractal hypergraphs mimics this concept for representing SystemC components.



The value associated to the identifier *id* the subhypergraph of the nearest hyperedge labelled by the identifier *id*, when we go back through the execution trace – *ie* when we go back on the hypergraph traversal. During this reverse, entering a lower hierarchical level is prohibited.

Figure 1: Embedding environment into fractal hypergraph

Notice that a lookup function is required. This function takes as argument an identifier and returns the expected result. If the identifier designates a variable, this function returns its value. If the identifier is associated to a function or a method, it returns the fractal hypergraph that represents its closure.

Definition 3 *A* system state *is represented by :*

- *t : the current time ;*
- *$\mathscr{H}_e$: the hypergraph that denotes the history of the system execution ;*
- *$\mathscr{H}_s$: the hypergraph representing the whole system.*

We define a transition function that maps a system state to another system state.

2.3.2 Some relevant semantics rules

The simplest case consists in the parallel execution of simple instructions.

Figure 2: Concurrent execution

Execution sometimes depends on a test, corresponding to the dispatch point pattern..

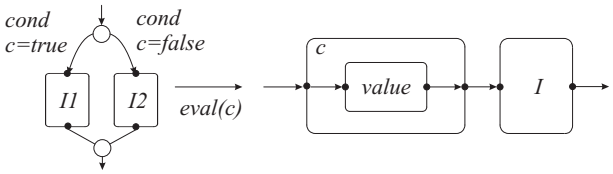


Figure 3: Conditional branching

2.3.3 For information

The whole construction of the C++ programming language in fractal hypergraphs has already been detailled in [11]. A semantics based on *fractal hypergraph* of a small behaviour on an intraprocedural language *MiniC* can be found here [12].

We detail the description of SystemC components and their synchronization through fractal hypergraphs.

3 Representing a SystemC component with Fractal Hypergraphs

The obvious mean to build a fractal hypergraph from a SystemC component is to parse the description into a fractal hypergraph, which represents the Control Flow Graph. In this part, we show the translation of some patterns into the corresponding fractal hypergraphs. Note that we will call the set of all possible execution paths Σ.

Sequences are represented by a sequence of hyperedges where each member is labelled by one of the terms of conjunction.

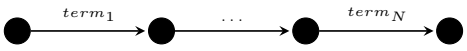


Figure 4: Sequence of instructions

$$\{s \in \Sigma \mid s = term_1 \ . \ * \ . \ term_N\}$$

Conditional dispatch points are useful to represent some conditional branching translated from instructions such as *if, switch*, etc.

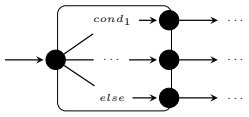


Figure 5: Conditional dispatch point

$$\{s \in \Sigma \mid s = cond_1.\sigma_1 \ \vee \cdots \vee \ \ s = \neg(cond_1 \vee \ldots).\sigma_{else}\}$$

Concurrent dispatch points are useful to represent the creation of several concurrent executions.

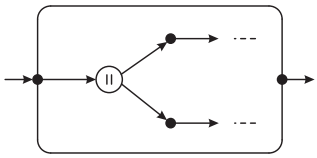


Figure 6: Concurrent dispatch point

Loops represent a sequence of iterative actions, such as *for* loops or *while* loops. These actions are very useful with automata.

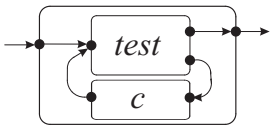


Figure 7: Conditional guarded loop

$$\{s \in \Sigma \mid s = \sigma_{init}.\sigma_{action}.(\neg test.\sigma_{latest}.\sigma_{action})*\} \cup$$
$$\{s \in \Sigma \mid s = \sigma_{init}.\sigma_{action}.(\neg test.\sigma_{latest}.\sigma_{action}) * .test.*\}$$

Synchronized hyperedges SystemC provides a "wait-/notify" mechanism which plays an important role in the SystemC scheduler [13] and serves as the basic engine of implementing interactions between processes. When modelizing the interactions between components, we should also modelize the interaction between processes and so modelize the "wait/notify" mechanism. *Synchronized hyperedges* denote two kind of hyperedges: hyperedges that wait for some event to be activated as well as hyperedges that activate those events.

Definition 4 *A hyperedge $\mathscr{E}$ is called a* standard hyperedge*, if all the awaited events and all the generated events can only be produced and consumed by synchronized hyperedges of its embedded fractal hypergraph.*

Definition 5 *A hyperedge $\mathscr{E}$ is called a* synchronized hyperedge*, if one of the following condition is verified :*

- *$\mathscr{E}$ may generate an event ;*
- *$\mathscr{E}$ may wait for an event ;*
- *there is at least one event generated by an embedded synchronized hyperedge that may be consumed by a synchronized hyperedege that is not located in its embedded hypergraph ;*
- *there is at least one event consumed by an embedded synchronized hyperedge that may be produced by a synchronized hyperedege that is not located in its embedded hypergraph ;*

Synchronized hyperedges or their embedded fractal hypergraph may also produce or consume events by communicating with other synchronized hyperedges.

A synchronized hyperedge $\mathscr{E}$ sends all the events that it may generate or that any of its sub-hyperedges may generate to all synchronized hyperedges that are waiting for these events and that belongs to the same fractal hypergraph as $\mathscr{E}$ and to its embedded fractal hypergraph. If a synchronized hyperegde $\mathscr{E}$ receives the notification of an event, it will propagate this event to all its synchronized sub-hyperegdes that are waiting for this event.

Objects extends data structures adding member functions (also called methods) to those structures. Member functions are easily represented by fractal hypergraphs.

Since the environment is embedded, fractal hypergraphs manage both environment and system representation.

Fractal hypergraphs provide the values of variables, the closures of functions and the execution traces in the meantime. An additional function will be necessary. This function takes as argument an identifier and must return the expected result. If the identifier corresponds to a variable, this function will return its value. If the identifier is associated to a function, it will return a fractal hypergraph representing the closure of the function.

Value hyperedges are hyperedges embedding a fractal hypergraph that defines a value.

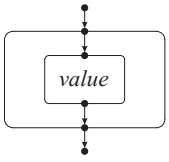


Figure 8: Value hyperedge

Data structures, such as arrays or records, can be defined as different variables embedded in a variable. Arrays need their index as identifiers, while records need their field names as identifiers.

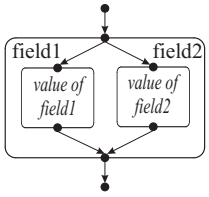


Figure 9: Date structure

In fractal hypergraphs defining a variable is just labelling a hyperedge, that embeddes its value, with the variable identifier.

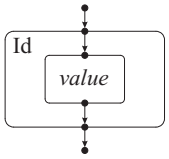


Figure 10: Variable location

Functions are represented by a fractal hypergraph made up of the function body and its bound arguments, represented by a fractal hypergraph. Each argument is represented by an empty hyperedge. The value of the arguments are represented by fractal hypergraphs.

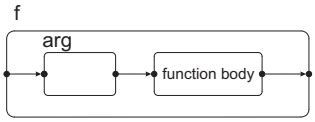


Figure 11: Function hyperedge

Lookup function must find the hyperedge labelled by its identifier. With data structures, the search function then enters into the hyperedge. By the same way, the search function will find the hyperedge of the embedded fractal hypergraph, that is labelled by the field identifier – field name for records, index for arrays, attributes or methods for objects.

Template is an abstract construction of the C++ langage, which enables to use generic programming. Templates can have value, class or simpler types in arguments. In SystemC, templates enable parametrizing components with values or other components. Template management consist of two steps : the template declaration and the template instantiation.

Step 1 – Template declaration
At beginning a template declaration, we create a fractal hypergraph in order to embed the template representation. It consists in an entry hyperedge, an exit hyperedge and a content hyperedge which aims at embedding the template representation. At this moment, the context hyperedge contains an empty subhypergraph.
Each times an instance of a template argument class is encountered, we check whether a new public method or public attribute is used. In this case, we add the external fractal hypergraph representing this public method – respectively attribute – to the content hyperedge.
At the end of this template declaration, we also have inferred the minimal interface that a template argument must respect to be used as an argument of this template. This minimal interface describes all public methods and attributes used in this template declaration.

Step 2 – Template instantiation
The first this template is instantiated with given arguments, a new fractal hypergraph is created to represent this instance. For each template argument, we select the hyperedges representing its used public methods and attributes. At this moment, these hyperedges only contains an empty subhypergraph. We also full them with their real content by duplicating the fractal hypergraphs representing these attributes or methods in the template argument – as the template processor does before compilation; see the dotted boxes on the following figure[1] :

```
template<class TValue> class Value
{
   public:
      TValue arg;

      void f1(int n) { /* ... */ }
      void fT(TValue t) { /* ... */ }
}
```

[1]Static members are not represented, but can easily be managed

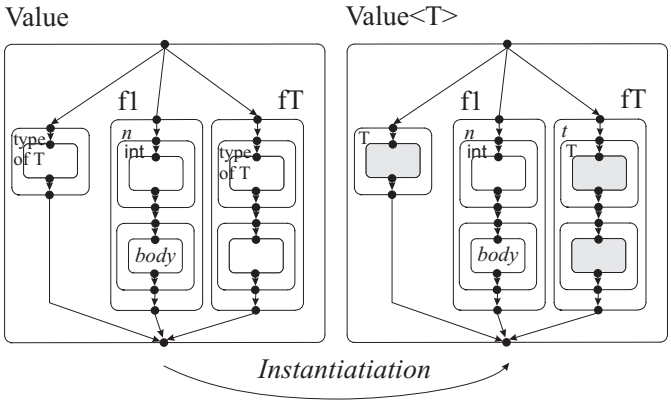


Figure 12: Template instantiation

Note Since we infer a fractal hypergraph representing the minimal interface of each template argument, using fractal hypergraph in a template processor enables checking contracts at this step of the compilation. This is a major difference with a C++ compiler whose template processor only does a syntaxtic work at template instantiation and lets the compiler checks for errors later. This method has a drawback : errors are detected after having modifying the original source code. The error messages are also quite difficult to read given the potential complex template nesting used in the source code. Since our method does not aim at only producing a new source code, we have to keep semantical information about templates during their construction. Our template management can also be used in compilation to provide clearer error messages when templates are used, as the *C++ concepts* wish.

4 Connecting Fractal Hypergraphs

In SystemC, components aggregates subcomponents. The aggregation of the components may be statically defined but can also be dynamically defined. However, connecting components is done dynamically. In the same way, associating components to signals and events is also done dynamically. In this section, we present how those operations impact fractal hypergraphs.

binding

```
/* ... */
com1.sub_com(com2
    );
/* ... */
```

In SystemC, this instruction binds the component *com*2 as the subcomponent called *sub_com* of the component *com*1.

Here you can see how a sub-component *sub_com* of the component *com*1 will be initialized with the component *com*2. At this moment, there are two ways to bind a component to another one. Binding is a method call in truth. Binding can also be an *external binding* or an *internal/inlined binding*.

external binding means that the subcomponent is acceded through a function call. When this subcomponent

is used, the processing first calls a function to access to the real component. Then it works with the component.
When building the fractal hypergraph representing the binded subcomponent, the processing generates wrappers to connect the internal subcomponent to the external real component. Each call of a subcomponent method goes through a hyperedge embedding a sequence of fractal hypergraphs representing the argument values. This hyperedge goes from the current hypernode to the first hypernode of the fractal hypergraphs representing the function body. Then the processing creates another hyperedge from each exit hypernode of the method body hypergraph to the hypernode when exiting function call hyperedge.

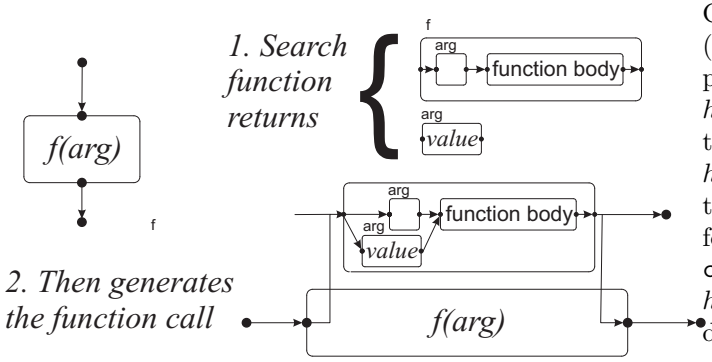


Figure 13: External function call

Given two fractal hypergraphs $h_1 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, {}^{\text{in}}\mathscr{V}_1, {}^{\text{out}}\mathscr{V}_1, \emptyset, \emptyset)$ for the uninitialized `com1.sub_com`, and $h_2 = (\mathscr{V}_2, \mathscr{H}_2, {}^{\text{in}}\mathscr{E}_2, {}^{\text{out}}\mathscr{E}_2, \mathscr{E}_2, {}^{\text{in}}\mathscr{V}_2, {}^{\text{out}}\mathscr{V}_2, {}^{\text{in}}\partial_2, {}^{\text{out}}\partial_2)$ for `com2`, we can define a fractal hypergraph for the final `com1.sub_com`, as $h = (\mathscr{V}_2, \mathscr{H}_2, {}^{\text{in}}\mathscr{E}_2, {}^{\text{out}}\mathscr{E}_2, \mathscr{E}_2, {}^{\text{in}}\mathscr{V}, {}^{\text{out}}\mathscr{V}, {}^{\text{in}}\partial, {}^{\text{out}}\partial)$ and :

- there exists a relation $entry \in \mathcal{P}({}^{\text{in}}\mathscr{V} \times {}^{\text{in}}\mathscr{V}_2)$
- there exists a relation $exit \in \mathcal{P}({}^{\text{out}}\mathscr{V} \times {}^{\text{out}}\mathscr{V}_2)$
- ${}^{\text{in}}\partial = \{(v_1, v_2) \in {}^{\text{in}}\partial_2 | \exists v \in {}^{\text{in}}\mathscr{E}_2,\ entry(v, v_1)\}$
- ${}^{\text{out}}\partial = \{(v_1, v_2) \in {}^{\text{out}}\partial_2 | \exists v \in {}^{\text{out}}\mathscr{E}_2, exit(v, v_1)\}$

internal binding means the methods to access the subcomponent are embedded into the main component.
Each call to a subcomponent method is substituted by a hyperedge boxing the body of this method. The fractal hypergraphs representing argument values are then embedded into the argument hyperedges. Finally execution trace records the method call and the processing enters the inlined hyperedge.

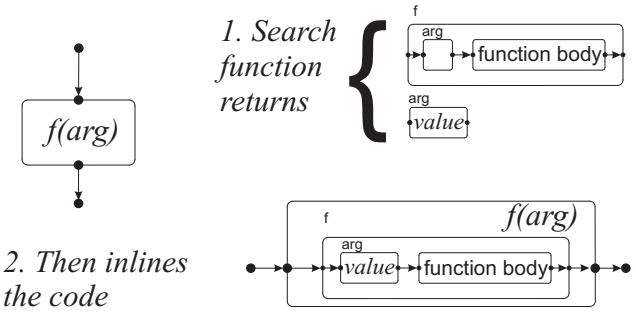


Figure 14: Inlined function call

Given three fractal hypergraphs $h_1 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, {}^{\text{in}}\mathscr{V}_1, {}^{\text{out}}\mathscr{V}_1, \emptyset, \emptyset)$ that denotes the wrappers that embed the method calls of `com1.sub_com`, $h_2 = (\mathscr{V}_2, \mathscr{H}_2, {}^{\text{in}}\mathscr{E}_2, {}^{\text{out}}\mathscr{E}_2, \mathscr{E}_2, {}^{\text{in}}\mathscr{V}_2, {}^{\text{out}}\mathscr{V}_2, {}^{\text{in}}\partial_2, {}^{\text{out}}\partial_2)$ that denotes the external real sub-component `com2`, and $h_p = (\mathscr{V}_p, \mathscr{H}_p, {}^{\text{in}}\mathscr{E}_p, {}^{\text{out}}\mathscr{E}_p, \mathscr{E}_p, {}^{\text{in}}\mathscr{V}_p, {}^{\text{out}}\mathscr{V}_p, {}^{\text{in}}\partial_p, {}^{\text{out}}\partial_p)$ that denotes the main component `com1`, we transform h_p into h'_p by integrating the description of `com2` directly inside the description of `com`. The final $h'_p = (\mathscr{V}'_p, \mathscr{H}'_p, {}^{\text{in}}\mathscr{E}'_p, {}^{\text{out}}\mathscr{E}'_p, \mathscr{E}'_p, {}^{\text{in}}\mathscr{V}_p, {}^{\text{out}}\mathscr{V}_p, {}^{\text{in}}\partial_p, {}^{\text{out}}\partial_p)$ is defined as follows :

- there exists a relation $entry \in \mathcal{P}({}^{\text{in}}\mathscr{V}_1 \times {}^{\text{in}}\mathscr{V}_2)$
- there exists a relation $exit \in \mathcal{P}({}^{\text{out}}\mathscr{V}_1 \times {}^{\text{out}}\mathscr{V}_2)$
- $\mathscr{V}'_p = \mathscr{V}'_p \cup \mathscr{V}_2$
- $\mathscr{H}'_p = \mathscr{H}_p \backslash h_1$
- ${}^{\text{in}}\mathscr{E}'_p = \{(o, d) \in {}^{\text{in}}\mathscr{E}_p \mid d \notin {}^{\text{in}}\partial_1\}$
- ${}^{\text{out}}\mathscr{E}'_p = \{(o, d) \in {}^{\text{out}}\mathscr{E}_p \mid o \notin {}^{\text{out}}\partial_1\}$
- $\mathscr{E}'_p = \mathscr{E}_p \cup \mathscr{E}_2 \cup \{(d, o) | \exists (n, n'), (o, n) \in {}^{\text{in}}\mathscr{E}_p \vee (n', d) \in {}^{\text{in}}\partial_2 \vee entry(n, n')\} \cup \{(d, o) \mid \exists (n, n'), (n, o) \in {}^{\text{out}}\mathscr{E}_p \vee (d, n') \in {}^{\text{out}}\partial_2 \vee exit(n, n')\}$

standard event association

```
/* ... */
SC_METHOD p <<
    clock;
/* ... */
```

The fractal hypergraph that represents the method p is embbeded in a synchronized hyperedge that consumes the *clock* event produced by the scheduler.

parallel event association

```
/* ... */
SC_THREAD p <<
    clock;
/* ... */
```

The fractal hypergraph that represents the method p is embbeded in a synchronized hyperedge that consumes the *clock* event produced by the scheduler. More generally the event may be produced by the global scheduler for global system generated events or by a synchronized hyperedge as seen in section 3.

5 Conclusion

This paper presents a new mathematical structure *fractal hypergraph* that can capture the object oriented nature of SystemC components. After providing the mathematical definition of fractal hypergraph and showing how we can define a formal storeless trace-based semantics based on fractal hypergraphs, we introduce how SystemC components may be represented using fractal hypergraphs; this includes classes, objects as well as template and template instantiation. We also show how the connection between SystemC components or the association of methods defined in a SystemC component to an event translate into this fractal hypergraph based representation.

This model based on fractal hypergaphs is successfully used in a SystemC formal debugger [4] and a complete formal SystemC semantics has been defined using fractal hypergraphs.

Since a hyperedge between the vertices of fractal hypergraphs may embed a fractal hypergraph, fractal hypergraphs can represent complex systems that mix program codes, hardware designs and logical formulae; each hyperedge of the hypergraph may be either a fractal hypergraph that represent a sequence of instructions (program representation), a sequence of logical operations (hardware representation), a logical formulae (specifications). We currently are working on static automatic analysis that incrementally replaces an implementation hyperedge with a logical hyperedge: *ie.* a hyperedge that describes logical formulae. We are also exploring an algorithm that checks if the implementation hyperedge verifies the constraints expressed by a logical hyperedge.

References

[1] F. Drewes, B. Hoffmann, and D. Plump, "Hierarchical graph transformation," *J. Comput. Syst. Sci.*, vol. 64, no. 2, pp. 249–283, 2002.

[2] C. Berge, *Graphs and Hypergraphs*. Elsevier Science Ltd, 1985.

[3] G. Gallo and M. G. Scutella", "Directed hypergraphs as a modelling paradigm," Tech. Rep., 1999.

[4] B. Monsuez, F. Védrine, and N. Vallée, "on the design of a formal debugger for system architecture," in *ICC'08: Proceedings of the 12th WSEAS international conference on Circuits*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 462–467.

[5] B. Monsuez, F. Védrine, M. Mayero, and N. Vallée, "How an "incoherent behavior" inside generic hardware component characterizes functional errors ?" in *CISST'09: Proceedings of the 3rd WSEAS international conference on Circuits, systems, signal and telecommunications*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2009.

[6] C. Colby and P. Lee, "Trace-based program analysis," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 195–207.

[7] D. A. Schmidt, "Trace-based abstract interpretation of operational semantics," *Lisp Symb. Comput.*, vol. 10, no. 3, pp. 237–271, 1998.

[8] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm, "A semantics for procedure local heaps and its abstractions," in *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2005, pp. 296–309.

[9] L. Liquori and M. L. Sapino, "Dealing with explicit exceptions in prolog," in *GULP-PRODE (2)*, 1994, pp. 296–308.

[10] S. Prasad and S. Arun-Kumar, "Introduction to operational semantics," in *The Compiler Design Handbook*, 2002, pp. 841–890.

[11] F. Védrine, "Analyses totales de programmes par interprétation abstraite, application au langage c++," Ph.D. dissertation, ENS Ulm, 1999.

[12] B. Monsuez, F. Védrine, and N. Vallée, "Creating an adaptative semantics upon fractal hypergraphs," in *waiting for publication*, 2010.

[13] B. Monsuez, Y. ZHANG, and F. Védrine, "Systemc waiting-state automata," in *VECoS'07*, Algiers, Algeria, May 2007.

[14] F. Nielson and H. R. Nielson, "Infinitary control flow analysis: a collecting semantics for closure analysis," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1997, pp. 332–345.

[15] R. Giacobazzi, ""optimal" collecting semantics for analysis in a hierarchy of logic program semantics," in *STACS '96: Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*. London, UK: Springer-Verlag, 1996, pp. 503–514.